Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 177 (2007) 219–233
www.elsevier.com/locate/entcs

Temporal Contextual Logic Programming

Vitor Nogueira[1]

Universidade de Évora and CENTRIA FCT/UNL
Portugal

Salvador Abreu[2]

Universidade de Évora and CENTRIA FCT/UNL
Portugal

Abstract

The importance of temporal representation and reasoning is well known not only in the database community but also in the artificial intelligence one. Contextual Logic Programming [17] (CxLP) is a simple and powerful language that extends logic programming with mechanisms for modularity. Recent work not only presented a revised specification of CxLP together with a new implementation for it but also explained how this language could be seen as a shift into the Object-Oriented Programming paradigm [2]. In this paper we propose a temporal extension of such language called Temporal Contextual Logic Programming. Such extension follows a reified approach to the temporal qualification, that besides the acknowledge increased expressiveness of reification allows us to capture the notion of *time of the context*. Together with the syntax of this language we also present its operational semantics and an application to the management of workflows.

Keywords: Temporal, logic, contexts, modular.

1 Introduction and Motivation

Contextual Logic Programming [17] (CxLP) is a simple and powerful language that extends logic programming with mechanisms for modularity. Recent work not only presented a revised specification of CxLP together with a new implementation for it but also explained how this language could be seen as a shift into the Object-Oriented Programming paradigm [2]. Finally, CxLP was shown to be a powerful language in which to design and implement Organizational Information Systems [3].

Temporal representation and reasoning is a central part of many Artificial Intelligence areas such as planning, scheduling and natural language understanding. Also in the database community we can see that this is a growing field of research [12,9].

[1] Email: vbn@di.uevora.pt
[2] Email: spa@di.uevora.pt

doi:10.1016/j.entcs.2007.01.025

Although both communities have several proposals for working with time, it still remains as challenging problem. For instance, there is no standard for temporal SQL or commercial DBMS that goes far beyond the *traditional* implementation of time.

To characterize a temporal reasoning system we can consider besides the ontology and theory of time, the "pure" temporal forms along with the logical forms. Although the first two issues are out of the scope of this paper, they were not neglected since they have been the subject of previous work [19,16], where we proposed a theory of time based on an ontology that considers points as the primitive units of time and the structure for the temporal domain was discrete and linear. Nevertheless, also intervals and durations were easily represented because the paradigm used to represent the temporal forms was Constraint Logic Programming. The last issue, the logical form of the reasoning system, is the subject of this work.

Adding a temporal dimension to CxLP results in a language that besides having all the expressiveness acknowledged to logic programming, allow us easily to establish connections to common sense notions because of its *contextual structure*. In this article we will introduce the language Temporal Contextual Logic Programming (TCxLP) along its operational semantics and discuss the application to the case of workflow management systems.

The remainder of this article is structured as follows. In Sects. 2 and 3 we briefly overview Contextual Logic Programming and Many–Sorted First–Order Logic, respectively. Section 4 discusses some temporal reasoning options followed and Sect. 5 presents the syntax and operational semantics of the proposed language, Temporal Contextual Logic Programming. Its application to the management of workflows is shown in Sect. 6. In Sect. 7 we establish some comparisons with similar formalisms. Conclusions and proposals for future work follow.

2 An Overview of Contextual Logic Programming

For this overview we assume that the reader is familiar with the basic notions of Logic Programming. Contextual Logic Programming (CxLP) [17] is a simple yet powerful language that extends logic programming with mechanisms for modularity. In CxLP a finite set of Horn clauses with a given name is designated by *unit*. The vocabulary of Contextual Logic Programming contains sets of variables, constants, function symbols and predicate symbols, from which terms and atoms are constructed as usual. Also part of the vocabulary is a set of unit names.

More formally, we have that each unit name is associated to a unit described by a pair $\langle L_u, C_u \rangle$ consisting of a label L_u and clauses C_u. The unit label L_u is a term $u(v_1, \ldots, v_n)$, $n \geq 0$, where u is the unit name and $v_1, \ldots, v_n$ are distinct variables denominated unit's *parameters*. We define a *unit designator* as any instance of a unit label.

In [2] we presented a new specification for CxLP, which emphasizes the OOP aspects by means of a stateful model, allowed by the introduction of unit arguments. Using the syntax of GNU Prolog/CX, consider a unit named `employee` to represent

some basic facts about University employees:

Example 2.1 *CxLP unit employee*

```
:-unit(employee(NAME, POSITION)).

name(NAME).
position(POSITION).

item :- employee(NAME, POSITION).
employee(bill, teachingAssistant).
employee(joe, associateProfessor).
```

The main difference between the code of example 2.1 and a regular logic program is the first line that declares the unit name (`employee`) along with two unit arguments (`NAME, POSITION`). Unit arguments help avoid the annoying proliferation of predicate arguments, which occur whenever a global structure needs to be passed around. A unit argument can be interpreted as a "unit global" variable, i.e. one which is shared by all clauses defined in the unit. Therefore, as soon as a unit argument gets instantiated, all the occurrences of that variable in the unit are replaced accordingly. For instance if the variable `NAME` gets instantiated with `bill` we can consider that the following changes occur:

```
:-unit(employee(bill, POSITION)).
name(bill).
item :- employee(bill, POSITION).
```

Consider another unit `baseSalary` that besides some facts has a rule to calculate the employees base salary: multiply an index by a factor that depends of the employee position. For instance, if the position is *teaching assistant*, then the base salary is $10(\text{index}) * 200(\text{factor}) = 2000$.

Example 2.2 *CxLP unit baseSalary*

```
:-unit(baseSalary(S)).

item :- position(P),
        position_factor(P, F),
        index(I),
        S is I*F.

position_factor(teachingAssistant, 200).
position_factor(associateProfessor, 400).

index(10).
```

We can see that there is no clause for predicate `position/1` in this unit. Although it will be explained in detail below, for now we can consider that the definition for this predicate will be obtained from the *context*.

A set of units is designated as a *contextual logic program*. With the units above we can build the program:

```
P = {employee, baseSalary}.
```

Since in the same program we could have two or more units with the same name but different arities, to be more precise besides the unit name we should also refer its arity i.e. the number of arguments. Nevertheless, most of the times when there is no ambiguity, we omit the arity of the units.

If we consider that `employee` and `baseSalary` designate sets of clauses, then the resulting program is given by the union of these sets.

For a given CxLP program, we can impose an order on its units, leading to the notion of *context*. Contexts are implemented as lists of unit designators and each computation has a notion of its *current context*. The program denoted by a particular context is the union of the predicates that are defined in each unit. Moreover, we resort to the *override semantics* to deal with multiple occurrences of a given predicate: only the topmost definition is visible.

To construct contexts, we have the *context extension* operation given by the operator `:>` . The goal U `:>` G extends the *current context* with unit `U` and resolves goal `G` in the new context. For instance to obtain the employees information we could do:

```
| ?- employee(N, P) :> item.

N = bill P = teachingAssistant ? ;
N = joe P = associateProfessor
```

In this query we extend the initial empty context `[]` [3] with unit employee obtaining context `[employee(N, P)]` and then resolve query `item`. This leads to the two solutions above.

Units can be stacked on top of a context; as an illustration consider the following query:

```
| ?- employee(bill, _) :> (item,
                           baseSalary(S) :> item).
```

In this goal we start by adding the unit `employee/2` to the empty context resulting in context `[employee(bill, _)]`. The first call to `item` matches the definition in unit `employee/2` and instantiates the remaining unit argument. The context then becomes

```
[employee(bill,teachingAssistant)].
```

After `baseSalary/1` being added, we are left with the context `[baseSalary(S), employee(bill,teachingAssistant)]`. The second `item/0` goal is evaluated and the first matching definition is found in unit `baseSalary`. In the body of the rule for `item` we find `position(P)` and since there is no rule for this goal in the current

[3] In the GNU Prolog/CX implementation the empty context contains all the standard Prolog predicates such as `=/2`.

unit (`baseSalary`), a search in the context is performed. Since `employee` is the topmost unit that has a rule for `position(P)`, this goal is resolved in the (reduced) context `[employee(bill, teachingAssistant)]`.

In an informal way, we can say that we ask the context for the position for which we want to calculate the base salary. Variable `P` is instantiated to `teachingAssistant` and computation of goal `position_factor(teachingAssistant, F), index(I), S is F*I` is executed in context `[baseSalary(S), employee(bill, teachingAssistant)]`. Using the clauses of unit `baseSalary` we get the final context `[baseSalary(2000), employee(bill, teachingAssistant)]` and the answer `S = 2000`.

3 Many–Sorted First–Order Logic

For self–containment reasons in this section we will briefly present Many–Sorted First–Order Logic (MSFOL). For a more detailed discussion see for instance [14].

Many–Sorted First-Order Logic can be regarded as a 'typed version' of First–Order Logic (FOL), that results from adding to the FOL the notion of sort. Although MSFOL is a flexible and convenient logic, it still preserves the properties of FOL.

In MSFOL besides predicate and function symbols, there exists sort symbols A, B, C. Each function f has an associated sort $sort(f)$ of the form $A_1 \times \ldots \times A_{arity(f)} \rightarrow A$ and each predicate symbol P has an associated sort $sort(P)$ of the form $A_1 \times \ldots \times A_{arity(P)}$. Likewise, each variable has an associated sort. These sorts have to be respected in order to build only well–formed formulas.

An MSFOL interpretation $\mathcal{M}$ consists of:

- a domain D_A for each sort A
- for each function symbol f a function $\mathcal{I}(f) : D_{A_1} \times \ldots \times D_{A_{arity(f)}} \rightarrow D_A$, matching the sort of f
- for each predicate symbol P a function $\mathcal{I}(P) : D_{A_1} \times \ldots \times D_{A_{arity(P)}} \rightarrow \mathcal{B}$, matching the sort of P.

The satisfiability for MSFOL interpretations is defined as expect, i.e. the quantification variables in clauses becomes sort–dependent.

4 Temporal Reasoning Issues

In this section we discuss several temporal reasoning options that we followed in our approach. Namely, the temporal qualification and temporal ontology.

4.1 The Model of Time

To define the model of time we need to define not only the time ontology but also the time topology. By time ontology we mean the class or classes of objects time is made of (instants, intervals, durations, ...) and the time topology is related to the

properties of sets of the objects defined, namely:

- discrete, dense or continuous
- bounded or unbounded
- linear, branching, circular or with a different structure
- are all individuals comparable by the order relation (connectedness)
- are all individuals equal (homogeneity)
- is it the same to look at one side or to the other (symmetry)

4.2 Temporal Qualification

Temporal qualification is by itself a very prolific field of research. For an overview on this subject see for instance [18]. Besides modal logic proposals, from a first–order point of view we can consider the following methods for temporal qualification: temporal arguments, token arguments, temporal reification and temporal token reification. Although no method is clearly superior to the others, we decided to follow a temporal reification to units designators. Because besides assigning a special status to time, temporal reification has the advantage of allowing to quantify over propositions. The major critics made to reification is that such approach requires a *sort structure* to distinguish between terms that denote real objects of the domain (terms of the original object language) and terms that denote propositional objects (propositions of the original object language).

One major issue in every temporal theory is deciding what sort of information is subject to change. In the case of Contextual Logic Programming the temporal qualification could be performed at the level of clauses, units or even contexts. In order to be as general as possible we decided to qualify units, more specifically, units designators. This way we can also qualify:

- clauses: by considering units with just one clause;
- contexts: by considering contexts containing a single unit.

Moreover, this way temporal qualification is twofolded: it is *static* when we are considering units and it is *dynamic* when those units are part of a context.

4.3 Temporal Ontology

From an ontological point of view we can classify the temporal relations into a number of classes such as fluents, events, etc. Normally, each of these classes has associated a theory of temporal incidence. For instance the occurrence of an *event* over an interval is *solid* (if it holds over a interval it does not hold on any interval that overlaps it) whereas *fluents* hold in a *homogeneous* way (if it holds in an interval then it holds in each part of the interval). Our theory of temporal incidence will be encoded by means of conditions in the operational semantics and to be as expressive as possible *unit designators* can be considered as events or as fluents according to the context, i.e. the current context will specify if they must hold in a solid or homogeneous way (see inference rule *Reduction* of page 10).

5 Temporal Contextual Logic Programming

In this section we present our temporal extension of CxLP, called Temporal Contextual Logic Programming (TCxLP). We start by providing the syntax of this language that can be regarded as a two–sorted CxLP where one of sort is for temporal elements and the other for non–temporal elements. Then we give the operational semantics by means of a set of inference rules which specify computations.

5.1 Syntax

Temporal Contextual Logic Programming is a two–sorted CxLP with the sorts $\mathcal{T}$ and $\mathcal{NT}$, where the former sort stands for temporal sort and the later for *non–temporal* sort. It is convenient to move to many–sorted logics since it naturally allows to distinguish between time and non–time individuals. There is a constant `now` of the temporal sort that stands for the *current time* and one new operator (`::`) to obtain the time of the context.

In Temporal Contextual Logic Programming each unit is described by a triple $\langle L_u, C_u, T_u \rangle$ where the element T_u is the temporal qualification. The unit temporal qualification T_u is a set of $holds(ud, t)$ where ud is a unit designator for u and t a term of the temporal sort.

To illustrate the concepts above, consider the table taken from [5] that represents information about Eastern Europe history, modeling the independence of various countries (to simplify we presented just the excerpt related to Poland) where each row represents an independent nation and its capital:

Year	**Timeslice**
1025	{ indep(Poland, Gniezno) }
...	
1039	{ indep(Poland, Gniezno) }
1040	{ indep(Poland, Cracow) }
...	

Table 1
Eastern European history: abstract temporal database

For this example we can consider the unit `indep` where the label is:

```
:- unit(indep(Country, Capital)).
```

the temporal qualification is:

```
holds(indep(poland, gniezno), time(1025, 1039)).
holds(indep(poland, cracow), time(1040, 1595)).
```

and the clauses are:

```
country(Country).
```

```
capital(Capital).
item :- holds( indep(Country, Capital) ).
```

A *temporal context* is any sequence of the following elements:

- unit designator,
- term of the sort $\mathcal{T}$.

With the unit above we can build temporal contexts like:

```
time(1400, 1600) :> indep(poland, C) :> item.
```

In this context `C` stands for name of all the capitals of (independent) Poland between 1400 and 1600.

In this section we are going to use λ to denote the empty context, t for a term of the temporal sort, *ud* for *unit designator*, $\overline{u}$ to represent the set of predicate symbols that are defined in unit u and C to denote contexts. Moreover, we may specify a context as $C = e.C'$ where e is the topmost element and C' is the remaining context (C' is the supercontext of C).

5.2 Operational Semantics

As usual in logic programming, we present the operational semantics by means of derivations. For self–containment reasons, we explain briefly what is a derivation. Such explanation follows closely the one in [15].

Derivations are defined in a a declarative style, by considering a derivation relation and introducing a set of inference rules for it. A tuple in the derivation relation is written as

$$\mathcal{U}, C \vdash G[\theta]$$

where $\mathcal{U}$ is a set of units, C a temporal context, G a goal and θ a substitution. Since the set of units remains the same for a derivation, we will omit $\mathcal{U}$ in the definition of the inference rules. Each inference rule has the following structure:

$$\frac{Antecedents}{Consequent}\,\{Conditions$$

The *Consequent* is a derivation tuple, the *Antecedents* are zero, one or two derivation tuples and *Conditions* are a set of arbitrary conditions. The inference rules can be interpreted in a declarative or operational way. In the declarative reading we say that the *Consequent* holds if the *Conditions* are true and the *Antecedents* hold. From a operational reading we get that if the *Conditions* are true, to obtain the *Consequent* we must establish the *Antecedents*. A derivation is a tree such that:

(i) any node is a derivation tuple

(ii) in all leaves the goal is null

(iii) the relation between any node and its children is that between the consequent and the antecedents of an instance of an inference rule

(iv) all clause variants mentioned in these rule instances introduce new variables different from each other and from those in the root.

The operation of the temporal contextual logic system is as follows: given a context C and a goal G the system will try to construct a derivation whose root is $C \vdash G\ [\theta]$, giving θ as the result substitution, if it succeeds. The substitution θ is called the *computed answer substitution*.

We may now enumerate the inference rules which specify computations. We will present just the rules for the basic operators since the remaining can be obtained from these ones: for instance, the extension `U :> G` can be obtained by combining the inquiry with the switch as in `:> C, [U|C] :< G`. Together with each rule we will also present its name and a corresponding number. Moreover, the paragraph after each rule gives an informal explanation of how it works.

Null goal

(1)
$$\frac{}{C \vdash \emptyset[\epsilon]}$$

The null goal is derivable in any context, with the *empty* substitution ϵ as result.

Conjunction of goals

(2)
$$\frac{C \vdash G_1[\theta] \qquad C\theta \vdash G_2\theta[\sigma]}{C \vdash G_1, G_2[\theta\sigma\lceil vars(G_1, G_2)]}$$

To derive the conjunction derive one conjunct first, and then the other in the same context with the given substitutions. The notation $\delta\lceil V$ stands for the restriction of the substitution δ to the variables in V.

Since C may contain variables in unit designators or temporal terms that may be bound by the substitution θ obtained from the derivation of G_1, we have that θ must also be applied to C in order to obtain the updated context in which to derive $G_2\theta$.

Context inquiry

(3)
$$\frac{}{C \vdash :> X[\theta]} \left\{ \theta = \mathrm{mgu}(X, C) \right.$$

In order to make the context switch operation (4) useful, there needs to be an operation which fetches the context. This rule recovers the current context C as a term and unifies it with term X, so that it may be used elsewhere in the program.

Context switch

(4)
$$\frac{C' \vdash G[\theta]}{C \vdash C' :< G[\theta]}$$

The purpose of this rule is to allow execution of a goal in an arbitrary context, independently of the current context. This rule causes goal G to be executed in

context C'.

Time inquiry: empty context

(5)
$$\frac{}{\lambda \vdash \ :: now[\epsilon]}$$

Time inquiry: temporal element

(6)
$$\frac{}{tC \vdash \ :: t'[\theta]} \begin{cases} sort(t) = \mathcal{T} \\ sort(t') = \mathcal{T} \\ \theta = mgu(t, t') \end{cases}$$

The two rules above state that the time of a context is *now* if the context is empty (5) or is given by the first element of the context, if such element is of the temporal sort (6). From the combination of these rules with the one for the context traversal (8) we get that the time of a context is represented by the "first" or topmost temporal element of such context (or *now* if there is no explicit mention of time). Therefore, also for the time enquiry operator we resort to an overriding semantics.

Reduction

(7)
$$\frac{uC\theta \vdash (G_1, G_2 \cdots G_n)\theta[\sigma]}{uC \vdash G[\theta\sigma\lceil vars(G)]} \begin{cases} H \leftarrow G_1, G_2 \cdots G_n \in u \\ \theta = \mathrm{mgu}(G, H) \\ holds(u\varphi, t') \in T_u \\ uC \vdash \ :: t \\ uC \vdash intersects(t, t') \end{cases}$$

The clauses of topmost unit (u) can be applied in a context (uC) if there is at least one unit designator ($u\varphi$) that holds ($holds(u\varphi, t')$) at the time of the context ($uC \vdash \ :: t$ and $uC \vdash intersects(t, t')$). In a informal way we can say that when a goal has a definition in the topmost unit in the context (first two conditions) and such unit (instantiation) can be applied in the time of the context (last three conditions), then it will be replaced by the body of the matching clause, after unification.

The reader might have noticed that not only the time (t) is obtained from the context but also the definition of predicate `intersects/2`. This way we can have not only different temporal elements (points, intervals, etc) but also different ontologies. For instance, if `t` and `t'` are time points and the definition of `intersects` in the context is a synonym for equal then we can consider unit designators as events. On the other hand if `t` and `t'` are intervals and the definition of `intersects` in the context is a synonym for intervals overlapping then we can consider unit designators as fluents. Finally, we can also have a combination of both (events and fluents) approaches.

Context traversal:

$$\frac{C \vdash G[\theta]}{eC \vdash G[\theta]} \quad (8)$$

When none of the previous rules applies, remove the top element of the context, i.e. resolve goal G in the supercontext.

5.2.1 Application of the rules

It is rather straightforward to check that the inference rules are mutually exclusive, leading to the fact that given a derivation tuple $C \vdash G[\theta]$ only one rule can be applied.

6 Application: Management of Workflow Systems

Workflow management systems (WfMS) are software systems that support the automatic execution of workflows. Although time is an important resource for them, the time management offered by most of these systems must be handled explicitly and is rather limited. Therefore, automatic management of temporal aspects of information is an important and growing field of research [8,6,7,13]. Such management can defined not only at the conceptual level (for instance changes defined over a schema) but also at run time (for instance workload balancing among agents).

The example used to illustrate the application of our language to workflows is based on the one given in [7]. The reason to use an existing example is two folded: not only we consider that such example is an excellent illustration of the temporal aspects in a WfMS but also will allow us to give a more precise comparison to the approach of those authors. For that consider the process of enrollment of graduate students applying for PhD candidate positions. In the first proposal of the process model, from September 1^{st}, 2003, any received application leads to an interview of the applicant (see workflow on the left of Fig. 1). After September 30^{th}, 2003, the process model was refined and the applicants CVs must be analyzed first: only applicants with an acceptable CV will be interviewed (see workflow on the right of Fig 1).

One process of the above workflow is selecting the successor(s) of a completed task. Since for the example given there is a refinement of the workflow process, such selection must depend of the time. For instance, if the completed task is "Receive Application" and the *current* date is the 4^{th} of September of 2003 then the next task must be "Interview". But if the current date is after September 30^{th}, 2003, then the next task must be "AnalyzeCV". To represent such process consider the following unit `next`. Please notice that besides the unit label and clauses, now we have the temporal qualification of the units designators. The first temporal qualification states that for the student enrolment the `next` task after receiving and application is doing an interview, but this is only valid between `'03-09-01'` and `'03-09-30'`.

```
% L_next: label
:- unit(next(SchemaName, TaskName, NextTask)).
```

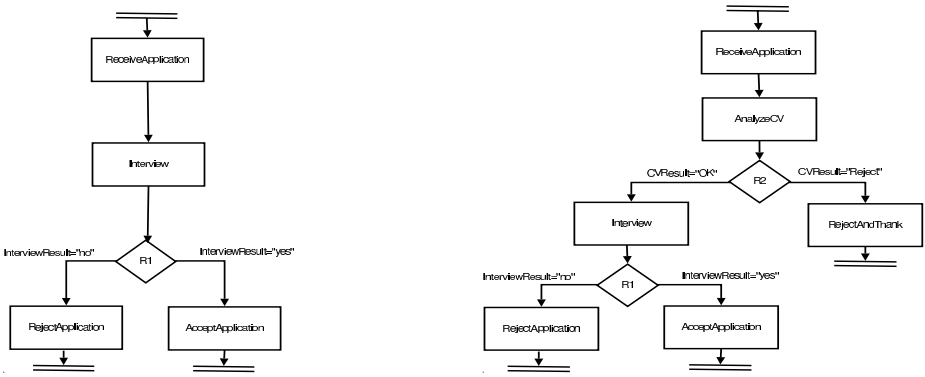

Fig. 1. The Student Enrolment process model: initial proposal (left) and refinement (right)

```
% C_next: clauses
item :- holds(next(SchemaName, TaskName, NextTask), _).

% T_next: temporal qualification
holds(next(studentEnrollment, receiveApplication, interview),
      time('03-09-01', '03-09-30')).
holds(next(studentEnrollment, interview, r1),
      time('03-09-01', inf)).
holds(next(studentEnrollment, rejectApplication, end_flow),
      time('03-09-01', inf)).
holds(next(studentEnrollment, acceptApplication, end_flow),
      time('03-09-01', inf)).
holds(next(studentEnrollment, rejectAndThank, end_flow),
      time('03-10-01', inf)).
holds(next(studentEnrollment, receiveApplication, analyzeCV),
      time('03-10-01', inf)).
holds(next(studentEnrollment, analyzeCV, r2),
      time('03-10-01', inf)).
```

With the unit above and assuming an homogeneous approach (see 4.3), consider the goal:

```
?- time('03-09-04') :> next(studentEnrollment, receiveApplication, N) :> item.
N = interview
```

I.e., at September 4^{th}, 2003, the next task after receiving an application is an interview. The same query could be done without the explicit time:

```
?- next(studentEnrollment, receiveApplication, N) :> item.
N = analyzeCV
```

Remembering that if nothing is said about the time then we assume we are in the present time (after September 30^{th}, 2003) and according to the refined workflow,

the next task must be to analyze the CV.

In the goals above our main focus was on the temporal aspects of the problem, leaving aside the modular aspects. Nevertheless we can consider a slightly more elaborated version of the problem where we have another temporally qualified unit called `worktask` with the name of the tasks and the role required by the agent for the execution: `unit(worktask(SchemaName, TaskName, Role))`. A variant of the query above that besides the next task (`N`) would also find out a (valid) role (`R`) for an agent to perform such task, could be stated as:

```
?- next(studentEnrollment, receiveApplication, N) :> (item,
        worktask(studentEnrollment, N, R) :> item).
N = analyzeCV
R = comitteeMember
```

7 Comparison With Other Approaches

One language that has some similarities with ours is Temporal Prolog of Hrycej [11]. Although such language provided a very interesting temporal extension of Prolog, it was restricted to Allen's temporal constraint model [4] for reasoning about time intervals and their relationships. Moreover, the predicate space was flat, i.e. it had no solution to the problem of modularity.

Its not a novelty the use of many–sorted logic to represent time (along other concepts). For instance to represent and reason about policies in [10] we find sorts for *principals*, *actions* and *time*. Moreover, there is an interesting notion of *environment* that can be regarded as a counterpart for *context*. Nevertheless since handling policies is the main focus, time is treated in a rather vague way.

Combi and Pozzi [8,6,7] have a very interesting framework for temporal workflow management systems. Their proposal is more "database oriented" and therefore presents the advantages and disadvantages known towards logical approaches. For instance, their queries are far more verbose (see for instance trigger *findSuccessor* in [7]), not only because we use logical variables but also because contexts allow us to make some facts *implicit*, i.e. in the context.

8 Conclusions and Future Work

In this paper we presented a temporal extension of CxLP that can be regarded as a two–sorted CxLP. Although we aimed that such extension could be as minimal as possible we also wanted to be as expressive as possible, leading to the notion of units whose applicability depends of the time of the context, i.e. temporally qualified units. Although we presented the operational semantics we consider that to obtain a more solid foundation there is still need for declarative approach together with its soundness and completeness proof.

To our understanding the best way to prove the usefulness of this language is by means of application, and for that purpose we chose the management of workflow systems. Besides this example, we are currently applying this language to the

legislation field, namely to represent and reason about the evolution of laws. As mentioned in Sect. 7, using TCxLP as a language for specifying policies seems as fruitfully area of research.

Finally, it is our goal to show that this language can act as the backbone for construction and maintenance of temporal information systems. Therefore, the evolution of this language or its integration with others such as ISCO [1] is one of the current lines of research.

References

[1] Salvador Abreu. Isco: A practical language for heterogeneous information system construction. In *Proceedings of INAP'01*, Tokyo, Japan, October 2001. INAP.

[2] Salvador Abreu and Daniel Diaz. Objective: In minimum context. In Catuscia Palamidessi, editor, *ICLP*, volume 2916 of *Lecture Notes in Computer Science*, pages 128–147. Springer, 2003.

[3] Salvador Abreu, Daniel Diaz, and Vitor Nogueira. Organizational information systems design and implementation with contextual constraint logic programming. In *IT Innovation in a Changing World – The 10th International Conference of European University Information Systems*, Ljubljana, Slovenia, June 2004.

[4] J .F. Allen. Maintaining knowledge about temporal intervals. *cacm*, 26(11):832–843, nov 1983.

[5] M. H. Boehlen, J. Chomicki, R. T. Snodgrass, and D. Toman. Querying TSQL2 databases with temporal logic. *Lecture Notes in Computer Science*, 1057:325–341, 1996.

[6] Carlo Combi and Giuseppe Pozzi. Temporal conceptual modelling of workflows. In Il-Yeol Song, Stephen W. Liddle, Tok Wang Ling, and Peter Scheuermann, editors, *ER*, volume 2813 of *Lecture Notes in Computer Science*, pages 59–76. Springer, 2003.

[7] Carlo Combi and Giuseppe Pozzi. Architectures for a temporal workflow management system. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 659–666, New York, NY, USA, 2004. ACM Press.

[8] Carlo Combi and Giuseppe Pozzi. Task scheduling for a temporal workflow management system. *Thirteenth International Symposium on Temporal Representation and Reasoning (TIME'06)*, 0:61–68, 2006.

[9] Chris Date and Hugn Darwen. *Temporal Data and the Relational Model*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[10] Joseph Y. Halpern and Vicky Weissman. Using first-order logic to reason about policies. In *16th IEEE Computer Security Foundations Workshop (CSFW-16 2003), 30 June - 2 July 2003, Pacific Grove, CA, USA*, pages 187–201. IEEE Computer Society, 2003.

[11] Tomas Hrycej. A temporal extension of prolog. *J. Log. Program.*, 15(1-2):113–145, 1993.

[12] Gad Ariav Ilsoo Ahn, Don Batory, James Clifford, Curtis E. Dyreson, Ramez Elmasri, Fabio Grandi, Christian S. Jensen, Wolfgang Käfer, Nick Kline, Krishna Kulkarni, T. Y. Cliff Leung, Nikos Lorentzos, John F. Roddick, Arie Segev, Michael D. Soo, and Suryanarayana M. Sripada. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.

[13] Elisabetta De Maria, Angelo Montanari, and Marco Zantoni. Checking workflow schemas with time constraints using timed automata. In Robert Meersman, Zahir Tari, Pilar Herrero, Gonzalo Méndez, Lawrence Cavedon, David Martin, Annika Hinze, George Buchanan, María S. Pérez, Víctor Robles, Jan Humble, Antonia Albani, Jan L. G. Dietz, Hervé Panetto, Monica Scannapieco, Terry A. Halpin, Peter Spyns, Johannes Maria Zaha, Esteban Zimányi, Emmanuel Stefanakis, Tharam S. Dillon, Ling Feng, Mustafa Jarrar, Jos Lehmann, Aldo de Moor, Erik Duval, and Lora Aroyo, editors, *OTM Workshops*, volume 3762 of *Lecture Notes in Computer Science*, pages 1–2. Springer, 2005.

[14] K. Meinke and J. V. Tucker, editors. *Many-sorted logic and its applications*. John Wiley & Sons, Inc., New York, NY, USA, 1993.

[15] Luís Monteiro and António Porto. A Language for Contextual Logic Programming. In K.R. Apt, J.W. de Bakker, and J.J.M.M. Rutten, editors, *Logic Programming Languages: Constraints, Functions and Objects*, pages 115–147. MIT Press, 1993.

[16] Vitor Beires Nogueira, Salvador Abreu, and Gabriel David. Towards temporal reasoning in constraint contextual logic programming. In *Proceedings of the 3rd International Workshop on Multiparadigm Constraint Programming Languages MultiCPL'04 associated to ICLP'04*, Saint–Malo, France, September 2004.

[17] António Porto and Luís Monteiro. Contextual logic programming. In Giorgio Levi and Maurizio Martelli, editors, *Proceedings 6th Intl. Conference on Logic Programming, Lisbon, Portugal , 19–23 June 1989*, pages 284–299. The MIT Press, Cambridge, MA, 1989.

[18] H. Reichgelt and L. Vila. *Handbook of Temporal Reasoning in Artificial Intelligence*, chapter Temporal Qualification in Artificial Intelligence. Foundations of Artificial Intelligence, 1. Elsevier Science, 2005.

[19] Vitor Nogueira. A Temporal Programming Language for Heterogeneous Information Systems. In G. Gupta M. Gabbrielli, editor, *Proceedings of the 21st Intl.Conf. on Logic Programming (ICLP'05)*, number 3668 in LNCS, pages 444–445, Sitges, Spain, October 2005. Springer.